DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

# Primer Trabajo Práctico

14 de Abril de 2010 — Algoritmos y Estructuras de Datos III


| Integrante | LU | Correo electrónico |
|---|---|---|
| Bianchi, Mariano | 92/08 | bianchi-mariano@hotmail.com |
| Brusco, Pablo | 527/08 | pablo.brusco@gmail.com |
| Di Pietro, Carlos Augusto Lyon | 126/08 | cdipietro@dc.uba.ar |





**Facultad de Ciencias Exactas y Naturales**
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

# Índice

# 1. Ejercicio 1

## 1.1. Introducción

El primer problema de este trabajo práctico consistió en dar un algoritmo que sea capaz de decir, dada una secuencia de números, la mínima cantidad de números a eliminar de la misma para que ésta sea unimodal. Esto es, que la secuencia cumple que es creciente desde el primer elemento hasta cierta posición y desde dicha posición es decreciente hasta el final[1].

Además, se estableció como requerimiento que la complejidad del algoritmo en cuestión fuera estrictamente menor que $O(n^3)$, siendo n el tamaño de la secuencia.

Se pensaron varias formas de encarar y resolver este problema, finalmente se aplicó la técnica de *programación dinámica*. En las sucesivas secciones, daremos una explicación detallada de cómo se implementó el algoritmo, cuál es su complejidad temporal y cómo se comportó frente a distintos valores y tamaños de entrada.

## 1.2. Explicación

Desde un comienzo se pensó que el problema podía resolverse usando *programación dinámica*, sin embargo comprender de qué forma debía aplicarse *principio de optimalidad* no fue, en principio, nada trivial. Tras un tiempo de analizar el problema, se pudo vislumbrar que el mismo podía ser visto como una variación de otro problema similar: el de encontrar dentro de una secuencia la subsecuencia creciente/decreciente más larga.
Básicamente, el razonamiento empleado fue el siguiente:

Si para cada posición $i$ de la secuencia se conoce:

- Longitud de la subsecuencia creciente más larga desde el principio hasta $i$ (que contenga al elemento i-ésimo)
- Longitud de la subsecuencia decreciente más larga desde $i$ hasta el final (que contenga al elemento i-ésimo)

entonces es posible decir cuál es el largo de una secuencia unimodal que tiene como "pivote" u objeto más grande al i-ésimo de la secuencia. Finalmente, sólo bastaría averiguar cuál es el "pivote" que maximice esa longitud para de ésta forma resolver el problema.
**Cabe observar, que determinar la longitud de la subsecuencia unimodal más larga es equivalente a encontrar la cantidad mínima de elementos a eliminar para transformar la secuencia dada en una secuencia unimodal. Este dado será tenido en cuenta a la hora de implementar el algoritmo.**

A continuación se desarrolla el funcionamiento del algoritmo mediante el cual se logra resolver el problema planteado.

Para construir el algoritmo de programación dinámica se definieron:

[1] Debe ser estrictamente creciente/decreciente

- un vector denominado *ascenso* donde se va guardando, en la i-ésima posicion, la longitud de la subsecuencia más larga desde el inicio de la secuencia original hasta la posición i. Esto es:
  $ascenso_i$ = longitud de la secuencia creciente más larga que termina con el número $v_i$
- la siguiente relación de recurrencia:
  - $ascenso_0 = 0$
  - $ascenso_i = max_{j<i}\{ascenso_j + 1\}$ (con $v_j < v_i$)
- y la solucion final: $max_{1 \leq i \leq n}\{ascenso_i\}$

Para entender el algoritmo principal, se presenta el pseudocódigo de la función *completar_ascensos*, la cual completa el vector *ascenso* con los valores correspondientes.

```
1 void completar_ascensos(ascenso: vector< nat >, v: vector< int >)
2 Complejidad: θ(tam(v)^2)
3 for (i = 0; i <= tam(v); i++);                    // θ(tam(v)^2))
4 do
5 |  ascenso[i] ← maximoAsc(ascenso,v,i) + 1;      // θ(i) ⊆ θ(tam(v))
6 end
```

La función maximoAsc(ascenso,v,i) indica, dentro del vector ascenso, la maxima longitud de un ascenso hasta la posicion i-1 del vector tal que cumpla la siguiente propiedad:

Si j es el indice del ascenso donde se encuentra el resultado, tiene que ocurrir que: [2]

$$(\forall k \in [0...i), v[k] < v[i])\ ascenso[j] > ascenso[k]$$

Por lo tanto, de esta manera podemos obtener para cada pivote, la longitud del mayor ascenso hasta esa posición.

Veamos un *ejemplo*:
$v = \{9, 5, 2, 8, 7, 3, 1, 6, 4\}$
Las subsecuencias ascendentes más largas son $\{2, 3, 4\}$ o $\{2, 3, 6\}$

| sucesión | 9 | 5 | 2 | 8 | 7 | 3 | 1 | 6 | 4 |
|---|---|---|---|---|---|---|---|---|---|
| longitud | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 3 | 3 |

Aquí podemos observar que en cada posición de la fila longitud, se encuentra la longitud de la secuencia creciente más larga hasta ese punto.

[2] ver pseudocódigo en el anexo

Una vez realizado este proceso, podemos obtener, ejecutando un algoritmo equivalente (pero desde el final hacia el principio) que obtenga, para cada posición, la longitud de la secuencia decreciente más larga desde la posición hacia el final. Al ejecutar esté proceso sobre el ejemplo obtendríamos:

| sucesión | 9 | 5 | 2 | 8 | 7 | 3 | 1 | 6 | 4 |
|---|---|---|---|---|---|---|---|---|---|
| ascendente | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 3 | 3 |
| descendente | 5 | 3 | 2 | 4 | 3 | 2 | 1 | 2 | 1 |

Luego, para decidir cual es el mejor pivot, nos basamos en dos filas extra implicitas que podemos ver a continuación :

| sucesión | 9 | 5 | 2 | 8 | 7 | 3 | 1 | 6 | 4 |
|---|---|---|---|---|---|---|---|---|---|
| ascendente | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 3 | 3 |
| descendente | 5 | 3 | 2 | 4 | 3 | 2 | 1 | 2 | 1 |
| suma | 5 | 3 | 2 | 5 | 4 | 3 | 1 | 4 | 3 |
| tam(S) - suma; | 4 | 6 | 7 | 4 | 5 | 6 | 8 | 5 | 6 |

Donde $suma_i = ascendente_i + descendente_i - 1 \ \forall i \in [1..tam(s)]$
(esto quiere decir, la suma de las longitudes de la secuencia de mayor longitud creciente hasta i + la longitud de la secuencia de mayor longitud decreciente desde i). Se le resta 1 porque ambas secuencias incluyen al elemento de la posición i.

La fila tam(S)-suma representa, cuantos elementos NO fueron usados para esta "escalera".

Por lo tanto se busca el máximo valor de la fila suma, lo que es equivalente a buscar el mínimo en la fila tam(s)-suma que representa la cantidad minima de borrados a realizar para lograr formar la secuencia unimodal y ese es el resultado devuelto por nuestra función.

### 1.3. Análisis de la complejidad del algoritmo

A continuación se calculará la complejidad del algortimo implementado en la función *escalerar*, que es el que resuelve el problema que se planteo. Primero veamos el pseudocódigo de la función completa:

void **escalerar**(v: vector⟨int⟩)

```
1  Complejidad: θ(tam(v)^2)
2  var max: nat;
3  var ascenso: vector⟨nat⟩(tam(v));                                   // θ(tam(v))
4  var descenso: vector⟨nat⟩(tam(v));                                  // θ(tam(v))
5  completar_ascensos(ascenso,v);                                      // θ(tam(v)^2)
6  completar_descensos(descenso,v);                                    // θ(tam(v)^2)
7  for (i = 0; i ≤ tam(v); i++) ;                                      // θ(tam(v))
8  do
9  |  if (ascenso[i] + descenso[i] ≥ max) then max ← ascenso[i] + descenso[i];
   |  // θ(1)
10 end
11 retornar tam(v) − max + 1;                                          // θ(1)
```

Pseudocódigo de la función completar_descensos:

void **completar_descensos**(descenso: vector⟨nat⟩, v: vector⟨int⟩)

```
1  Complejidad: θ(tam(v)^2)
2  for (i = tam(v); i ≥ 0; i−−) ;                                      // θ(tam(v)^2)
3  do
4  |  descenso[i] ← maximoDes(descenso,v,i) + 1;    // θ(tam(v) − i) ⊆ θ(tam(v))
5  end
```

Pseudocódigo de la función maximoAsc:

nat **maximoAsc**(ascenso: vector⟨nat⟩, v: vector⟨int⟩,i: nat)

```
1  Complejidad: θ(i)
2  var res: nat ;                                                      // θ(1)
3  res ← 0 ;                                                           // θ(1)
4  for (k = 0; k < i; k++);                                            // θ(i)
5  do
6  |  if (v[j] < v[i] and res < ascenso[j]) then res←ascenso[j];       // θ(1)
7  end
```

Aclaración: el pseudocódigo de la función competar_ascensos() se encuentra en la sección "Explicación" y la función maximoDes() es analoga a maximoAsc().

Como se ve en los pseudocódigos, sin depender de los valores ingresados en el vector v, la cantidad de operaciones es la misma para un mismo tamaño de entrada. Por lo tanto, podemos decir que esta función esta acotada por arriba y por abajo por una función cuadratica a partir de un $n_0$ es decir, escalerar() $\in \Theta(tam(v)^2)$

En conclusión, nuestro algoritmo esta siempre acotado por arriba y por abajo por una función cuadratica en el tamaño de la entrada (tam(v)).

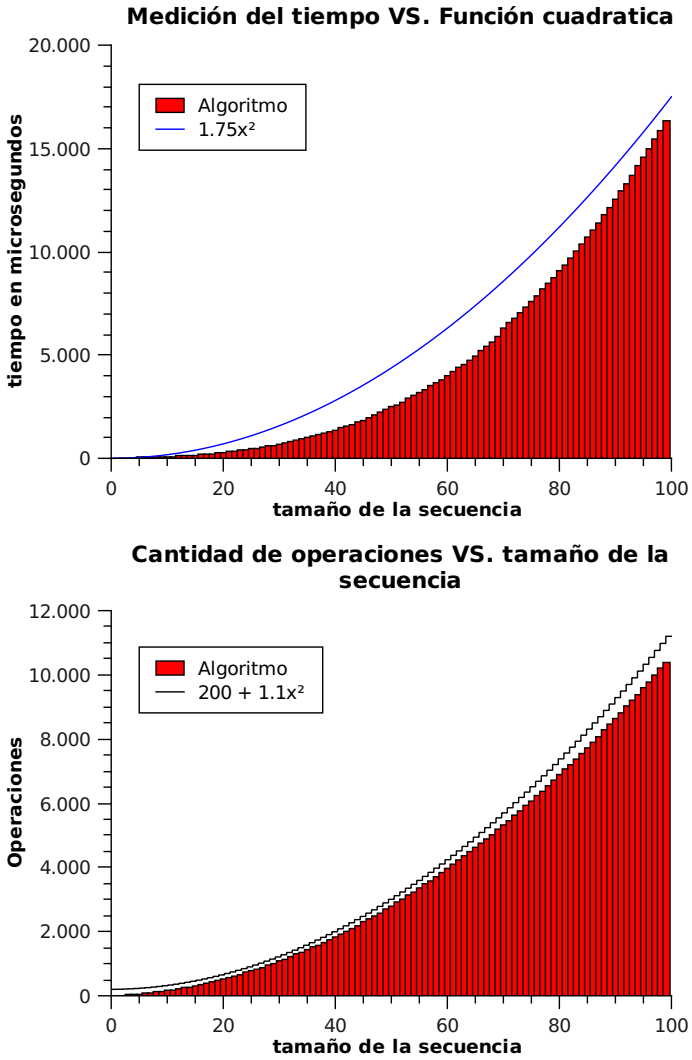
### 1.4. Detalles de implementación

Para compilar el programa sólo hace falta ejecutar el comando make en consola.

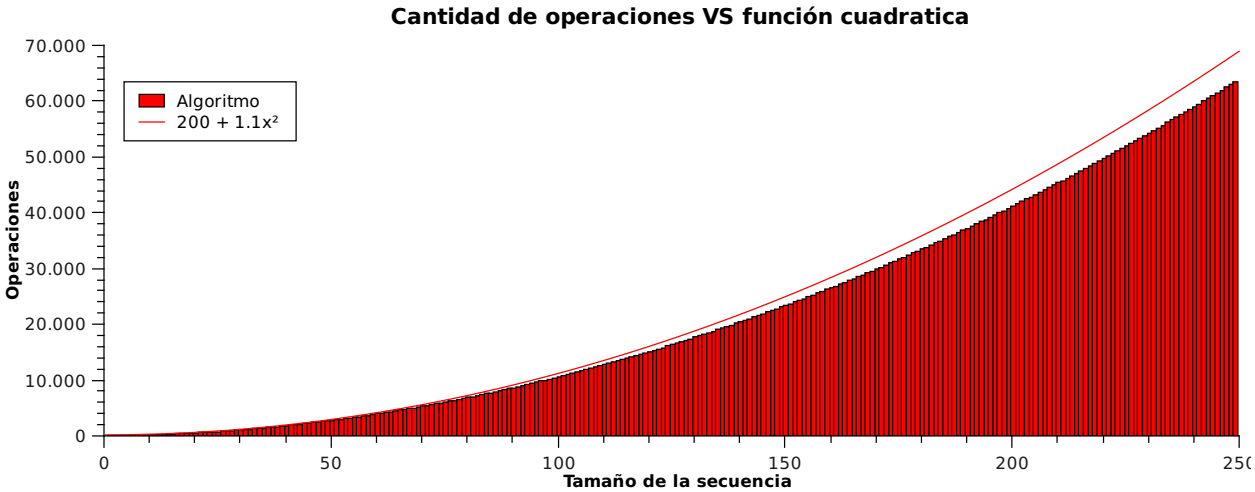
**Modo de Uso**

## 1.5. Resultados

A continuación veremos gráficos que muestran el comportamiento del algoritmo utilizado




Cuadro 1: Estos gráficos muestran el tiempo de ejecución y cantidad de operaciones en secuencias de 1 hasta 100 elementos comparados con funciones cuadraticas




Este gráfico muestra la cantidad de operaciones en secuencias de 1 hasta 250 elementos comparados con una funcion cuadratica (notar que es la misma función cuadratica que en el gráfico anterior

## 1.6. Debate

## 1.7. Conclusiones

# 2. Ejercicio 2

## 2.1. Introducción

## 2.2. Explicación

Primero, algunas definiciones:

**Def$_1$:**
Un grafo $G$ es fuertemente orientable si existe una asignación de direcciones a los ejes del conjunto de ejes del grafo $G$ tal que el digrafo resultante es fuertemente conexo.

**Def$_2$:**
un puente, **arista de corte** o istmo es una arista que al eliminarse de un grafo incrementa el número de componentes conexos de éste. Equivalentemente, una arista es un puente si y sólo si no está contenido en ningún ciclo.

**Teorema** [Robbins, 1939] :
Un grafo conexo $G$ es fuertemente orientable si y solo si $G$ no tiene puentes (Demostración en el anexo).

Con esté teorema podemos ver que si encontramos al menos 1 puente en nuestro grafo, significa que no podremos orientarlo como queremos, y de lo contrario, si encontramos que no hay ningun puente, podremos orientarlo. Por lo tanto, el algoritmo utilizado realiza exactamente esa comprobación. Veamos como trabaja:

**comprobación(Grafo G)**

```
1 Complejidad: O(n^3)
2 var eje : int ← 0
3 var n : int ← cantNodos(G)
4 while eje ≤ m do
5 |   k ← RecorridoSinEje(eje,G)
6 |   if k ≠ n then return no se puede
7 |   eje ← eje + 1
8 end
9 return fuertemente conexo
```

RecorridoSinEje(eje,G) es una funcion que recorre el grafo $G$ (con BFS o DFS) sin utilizar la arista $eje$ y retorna la cantidad de nodos visitados, $k$. Como la forma de recorrer utilizada, solo recorre nodos conectados a la raiz (es decir, al nodo donde comienza el recorrido), quiere decir que el resultado k va a ser n (la cantidad total de nodos de $G$) si $G$ es conexo. De lo

contrario, si k es menor que $n$, estamos en presencia de 2 componentes conexas (o más en el caso de $eje = 0$). Por lo tanto, el eje sacado, era un puente.

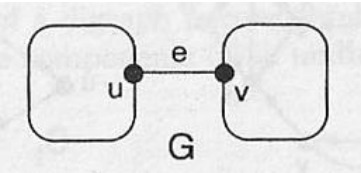
Aclaración:
Cuando eje = 0, representa, recorrer a G completo con todos sus nodos. En este punto podria pasar que G no sea conexo y esta función devolvería el resultado correspondiente.

## 2.3. Anexo: Demostraciones

**Teorema** [Robbins, 1939]:
Un grafo conexo $G$ es fuertemente orientable si y solo si $G$ no tiene puentes.

**Demostración: $\rightarrow$)**

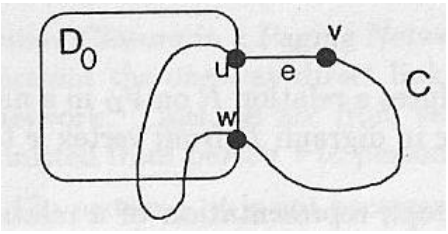
Utilizando el contrareciproco, supongamos que el grafo $G$ tiene una arista de corte $e$ que une a los vertices $u$ y $v$. Entonces el unico camino entre $u$ y $v$ o $v$ y $u$ en el grafo $G$ es $e$ (ver figura). Por lo tanto para cualquier asignacion de direcciones, el nodo cola($e$) nunca va a poder ser alcanzada por el nodo cabeza($e$) (ver cuadro 1).




Cuadro 2:

**Demostración: $\leftarrow$)**

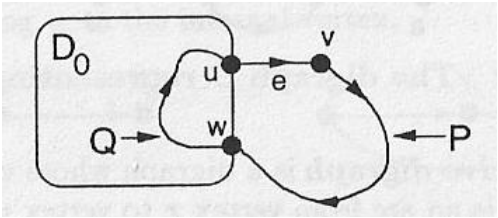
Supongamos que $G$ es un nodo conexo sin aristas de corte. Por esto toda arista en $G$ cae en un circuito de $G$. Para hacer que $G$ sea fuertemente orientable, empezaremos con cualquier circuito ($D_0$) de $G$ y dirigiremos sus ejes en una dirección (obteniendo un circuito dirigido). Si el ciclo $D_0$ contiene todos los nodos de $G$, entonces la orientación esta completa (ya que $D_0$ es fuertemente conexo). De lo contrario, hay que elegir cualquier eje $e$ uniendo a un vertice $u$ en $D_0$ y a un vertice $v$ en $V_G - V_{D_0}$ (ese eje existe ya que $G$ es conexo). Sea C = $\langle u, e, v, ..., u \rangle$ un ciclo que contiene al eje $e$, y sea $w$ el primer vertice luego de v en $C$ que cae en el ciclo $D_0$ (ver cuadro 2).




Cuadro 3:

A continuación, direccionamos los ejes de este camino entre $v$ a $w$, obteniendo el camino dirigido $v-w$ que llamaremos $P$. Luego direccionamos el eje $e$ desde $u$ hasta $v$ y consideramos el digrafo $D_1$ que se forma agregando el eje dirigido $e$ a $D_0$ y todos los vertices y ejes dirigidos del camino $P$. Como $D_0$ es fuertemente conexo, entonces hay un camino dirigido de $w$ a $u$ ($Q$) en $D_0$ (ver cuadro 3). La concatenación de $P$ con $Q$ y el eje dirigido $e$ forman un circuito simple direccionado que contiene $u$ y los nuevos vertices de $D_1$. (si los vertices $u$ y $w$ son el mismo, entonces $P$ satiface ser un circuito simple dirigido)

Por lo tanto, el vertice $u$ y todos estos nuevos vetices son mutuamente alcanzables en $D_1$ y además $u$ y cada vectice del digrafo $D_0$ tambien son mutuamente alcanzables, y, por lo tanto , el digrafo $D_1$ es fuertemente conexo. Este proceso puede continuar hasta que el digrafo $D_1$ para algun $l \geq 1$, contenga todos los vertices de $G$. En este punto, cualquier asignacion de direcciones hacia las aristas sin direccion restantes completaran la orientación de $G$, puesto que contendrá el digrafo fuertemente conexo $D_l$ como subdigrafo.




Cuadro 4: